IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance

Course code: KSDSESM1KU

MSc in Computer Science

DevOps: ITU-MiniTwit

Group E — Grl Pwr

IT University of Copenhagen

Name	Email
Amalie Bøgild Sørensen	abso@itu.dk
Andreas Nicolaj Tietgen	anti@itu.dk
Malin Birgitta Linnea Nordqvist	bino@itu.dk
Mille Mei Zhen Loo	milo@itu.dk
My Marie Nordal Jensen	myje@itu.dk
Sarah Cecilie Chytræus Christiansen	sacc@itu.dk

May 10, 2024

2500 words

Table of Contents

1 System's Perspective

1.1 Design and Architecture

Figure 1 provides an overview of our system infrastructure:

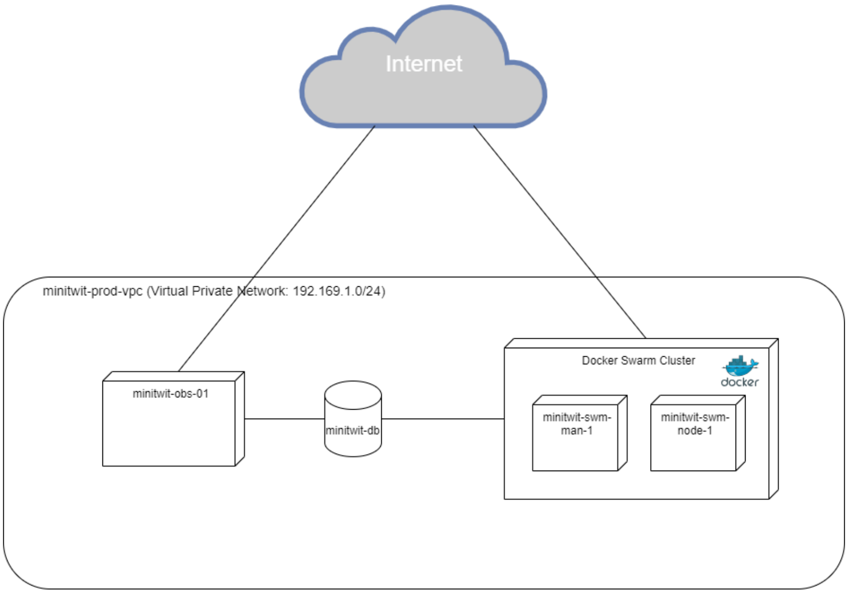

Figure 1: *Overview of the system*

The system includes a single Virtual Private Network with an IP 192.168.1.0/24. Within this network, we operate a database cluster consisting of a single node. This node is connected to the Docker Swarm cluster, featuring both a manager and a worker node.

Both our monitoring and cluster servers are visibly exposed to the internet. Additionally, the database server is accessible from the internet but has constraints to only talk to our servers and therefore not accept requests from any other host.

1.2 Interactions of Subsystems

Our system consists of three subsystems, namely SimulatorAPI, Minitwit, and Minitwit.Infrastructure. The SimulatorAPI and Minitwit subsystems, as can be seen in figure 2, are similar in structure, but differ since SimulatorAPI is a REST service and Minitwit is a website. Both controllers use the package in the subsystem Minitwit.Infrastructure to query for data using the Repository pattern.

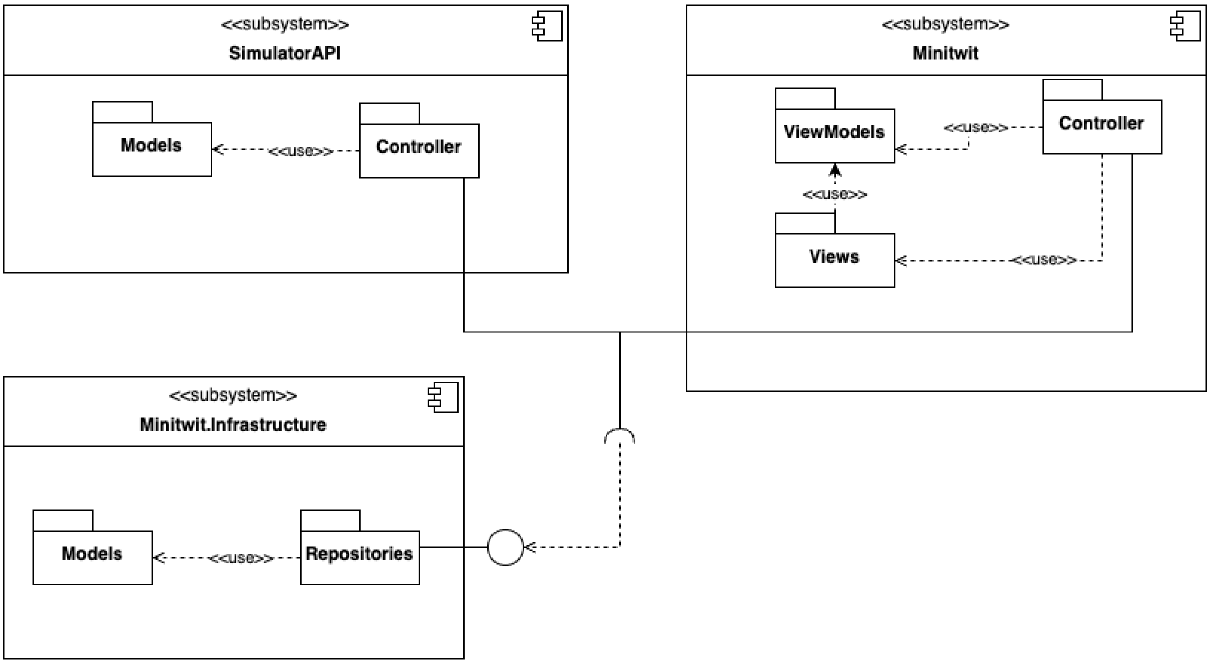

Figure 2: *An overview of the subsystems and how they interact*

As can be seen in figure 3 the controller for Minitwit handles a lot of logic in order to present the correct information to the user. However for SimulatorAPI, in figure 4 there is little to non logic, and it only calls the repository to fetch the user messages.

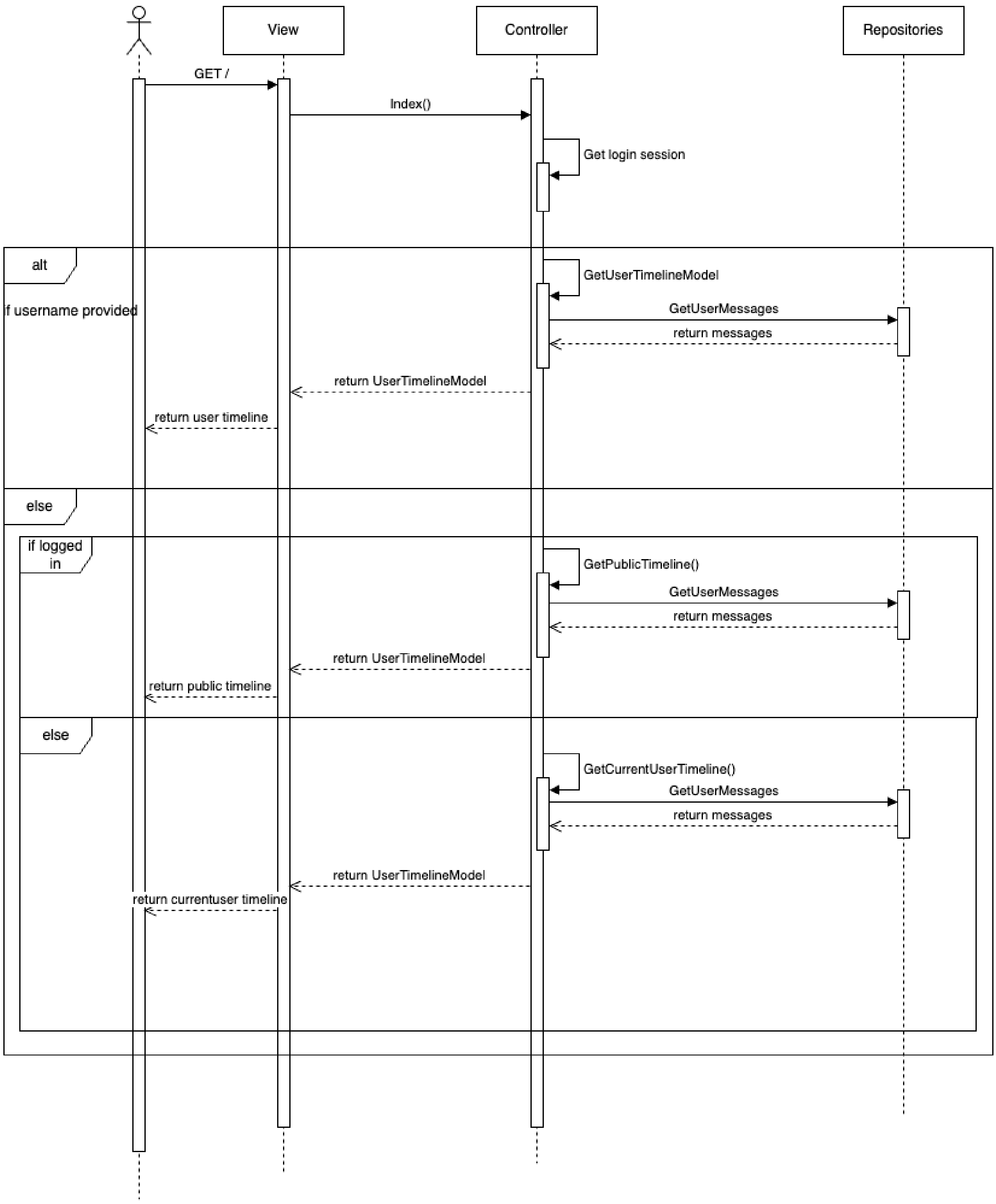

Figure 3: *Minitwit: Sequence diagram of a user call to the index page*

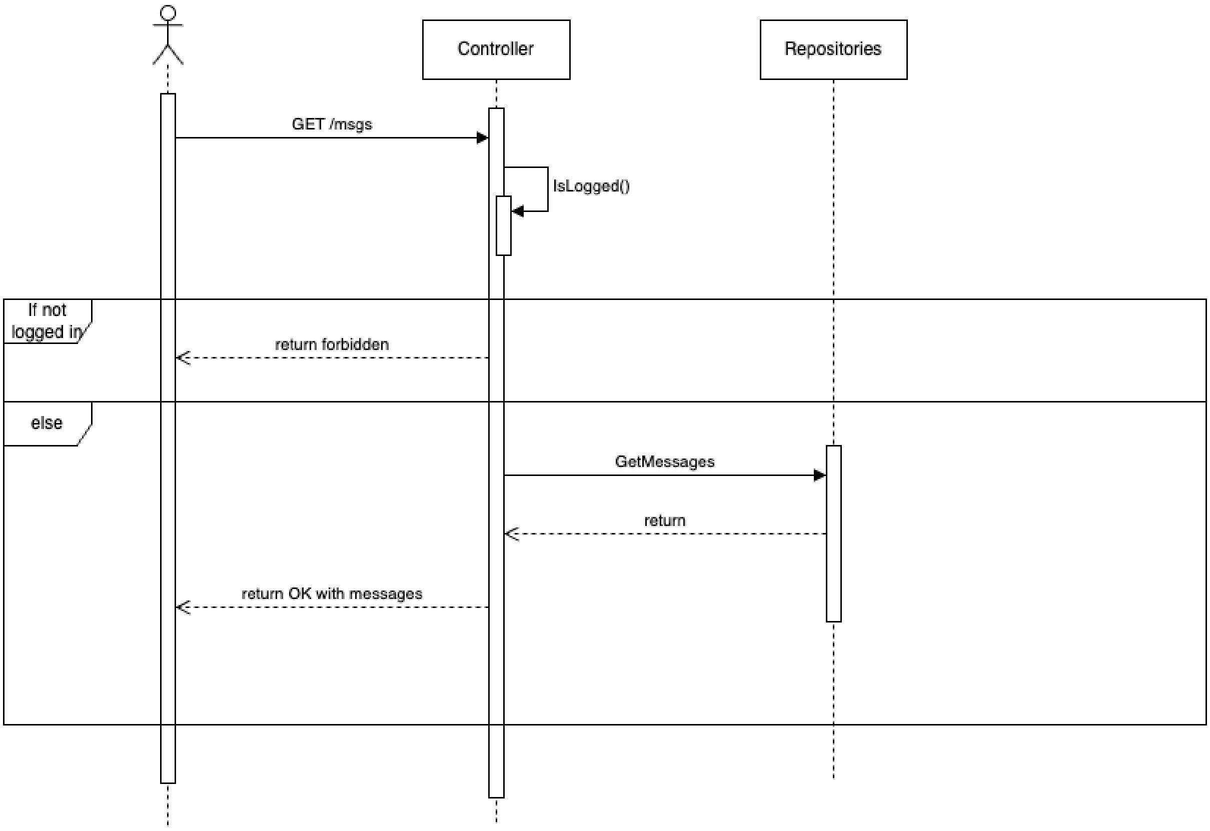

Figure 4: *SimulatorAPI: Sequence diagram of a user call to get messages*

1.2.1 Minitwit.Infrastructure

In the previous section we saw that the controller logic is very different but needs to call the same queries. Minitwit.Infrastructure is implemented to share duplicate code between the SimulatorAPI and Minitwit. This can be seen with the interaction of the package Repositories which leverages the Repository Pattern to abstract the logic of handling database queries and make it testable.

1.3 Current state

Table 1 describes the status of the system after the simulator has been shut down.

Total requests	34233
Requests secured	34233
Total unhandled exceptions	18
P99 min response time	94.2 ms
P99 max response time	716 ms

Table 1: *System metrics*

Furthermore, the static analysis tool Sonarcloud assesses that the quality of the overall code is good (see appendix B).

2 Process Perspective

2.1 Monitoring and logging

For monitoring we use Grafana. We export metrics by using OpenTelemetry and collect the metrics via Prometheus and lastly push them to Grafana. Our board shows requests duration, errors rate, top 10 unhandled exception endpoints and more (for visualization of the board, see appendix A). We have used the board to get an overview of where to put our focus, ie. which endpoints to improve, which errors to fix, etc. Moreover, we are gaining insights into the health of the system and get an impression of how the backend and frontend handle the requests.

For logging, we use Grafana and Loki. It seemed natural to continue our work with Grafana in order to keep the system setup as simple as possible. The logs are divided into Information, Warning, Debug and Error. All logging statements are placed in the controllers, such that we have information about the users' whereabouts in the system. For example, we log when a user logins whether successful or not.

2.2 Security Assessment

Based on our security assessment (see appendix E), we constructed the following risk matrix:

	Rare	Unlikely	Possible	Likely	Certain
Catastrophic	Exposed Secrets		Depricated Dependencies		Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing		
Marginal				Log Injection	
Negligible					
Insignificant					

Figure 5: *Risk matrix from security assessment*

We decided to focus on the scenarios in the red zone, as they would be the cause of most damage. Due to the team not having enough knowledge about how the simulator worked in regards to creating users and logging in, we did not move further with the *Password Brute Forcing* scenario. However, possible solutions could be to incorporate 2FA, password requirements and time-out limitation for an actual system in production.

For *Log Injection*, we made sure to sanitize user inputs, as they previously were put directly into the log, which could be exploited to hide ill-intentioned actions. We thereby hardened the system against injection attacks.

With *Depricated Dependencies*, we chose to integrate the tool *dependabot* into our CI/CD pipeline to watch out for outdated dependencies in our main repository. This helps us ensure that an adversary is not able to take advantage of the team not being aware of vulnerabilities in variuos of the used tools.

Open Ports was deemed the biggest risk, as this was most likely the reason for us being hacked, which we have written more about in section 3.1 about being hacked. We used the tool *nmap* to check for open ports, where we then tried to close the open port used for Prometheus. It was unclear if we were successful as *nmap* kept showing the port as open despite seeing the firewall configuration saying otherwise on the server itself. We also attempted to switch from HTTP to HTTPS, as it would strengthen the overall security of our application, but here we ran into several bureaucratic issues with changing the name servers for the domain, which resulted in us running out of time for the task. Ideally, this would have been accomplished to further harden our system's security.

2.3 Scaling

We use horizontal scaling, as we wanted a more resilient application and limit downtime when deploying. We settled on Docker Swarm over building a custom load balancer or adopting Kubernetes, primarily due to its simplicity and compatibility with the existing system setup. To implement Docker swarm we created a new server in our system and added a manager and a worker node to the swarm.

When attempting to scale by adding another frontend replica, we encountered challenges. In order for the two replicas to work together we needed to handle distributed sessions to remember who is logged in even though the frontend replica is switched. However, we did not succeed in this (we refer to week 11 in the log in appendix D), which is the reason why we only have a single frontend replica.

For the update strategy we went with rolling updates because it is the default update strategy for Docker Swarm.

3 Lessons Learned

3.1 Lesson 1: Getting Hacked

Just a couple of hours after attending the lecture on security, we got hacked, leading us to experience firsthand how important it is to incorporate security in a CI/CD pipeline.

The first suspicion we got was when we discovered, through our monitoring, that the response time of our server was suddenly very slow. This led us to our Digital Ocean dashboard which showed that the server was using 100% CPU power, which is highly unusual.

From that point the group scoured the server for clues as to what was happening, finding countless calls to `masscan` essentially drowning our server, as well as mysterious installations and what looked like a call to a remote script via a cronjob.

After a few failed attempts to evict the adversary it was decided to destroy the server, as we fortunately had already implemented our Infrastructure as Code, so provisioning and deploying a new server could be done in under half an hour.

After some introspection into our system, we assume that the adversary had gotten access via some open ports that we were unaware were exposed. The ports became exposed in an attempt to make the network function between servers in a docker swarm, which seems to override the firewall.

Learning from this, we have worked to close exposed ports from Docker and finding alternative solutions to setting up the network. Another key takeaway is that because we had the necessary monitoring in place, to figure out that the server was being targeted, as well as having implemented Infrastructure as Code, we were able to detect and react to the attack fairly quickly, giving us only a few hours of downtime.

3.2 Lesson 2: Shift from Vagrant to Ansible-Pulumi

Another lesson we have learned during this project is the importance of choosing the right tools for a project. At the start of the project, we had chosen to provision our VMs with Vagrant, inspired by the exercises from the course. When realizing later in the project that we would have to switch Digital Ocean account at some point due to running out of credit, we had to streamline the setup of our VMs.

The choice of Configuration Management tool fell upon Ansible, which was supposed to call the Vagrant files from a config server, provisioning the web and monitoring servers. However, when having more complicated automation and collaboration needs for our project, it turned out that Vagrant was not the right tool for the job. After many hours of attempting to get Vagrant to work with Ansible, we found out that Vagrant saves local metadata to maintain some state, which was making the provisioning from Ansible and the config server fail[2]. Furthermore, Vagrant does not seem to be supported with GitHub Actions - possibly also due to how it handles state. We decided to cut our losses with Vagrant and search for a more suitable tool that could help us write our infrastructure as code. The choice fell upon Pulumi in the end. It was a valuable lesson to see how it made a difference when we took the time to investigate different tools and their properties so we could make an informed decision based on the knowledge of our system's needs.

3.3 DevOps Style

When reflecting on how we as a group incorporated the style of DevOps into our way of working, we recalled the *Three Ways*, which were the characterising principles for processes and behaviour in DevOps, that consisted of *Flow*, *Feedback* and *Continual Learning and Experimentation*[3].

In regards to the principle of flow, it wasn't hard to adopt the ideas of making our work visible and reducing batch sizes, as all members of the group have previously worked agile in other courses, which also embodies those same ideas. Using a kanban board to track our tasks and their progress not only helped us stay on target, but also helped us with confining each task such that it could be deployed continuously[3].

For the principle of feedback, the concept of peer reviewing via pull requests on GitHub helped install a sense of ownership over the application. Automating the process of not only testing, but also building and deploying the entire application in a continous fashion also allowed for errors to be found and mitigated quickly compared to earlier projects we have worked on, where it was easy for the issues to pile up on each other.

Lastly, with the principle of continual learning and experimentation, having a safe system of work[3] was crucial for us to learn and grow in a secure environment, which in turn allowed for greater experimentation in how to improve the system and its setup. By keeping weekly work logs (see appendix D) and guides easily available, each member would have the same oppertunity to gain a deeper and better understanding of the system as a whole.

4 References

[1] Ansible. *Docker Swarm Module*. 2024. URL: `https://docs.ansible.com/ansible/latest/collections/community/docker/docker_swarm_module.html`.

[2] DevOps Group e. *Github Issue: Add playbooks in the vagrant provisioning steps*. 2024. URL: `https://github.com/devops2024-group-e/itu-minitwit/issues/178`.

[3] Gene Kim. *The DevOPS Handbook - How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. It Revolution Press, 2016. ISBN: 9781942788003.

[4] OWASP. *OWASP Top Ten*. 2021. URL: `https://owasp.org/www-project-top-ten/`.

Appendices

Table of Contents

A Grafana

Figure 7: *A snapshot of the Grafana dashboard.*

B Sonarcloud

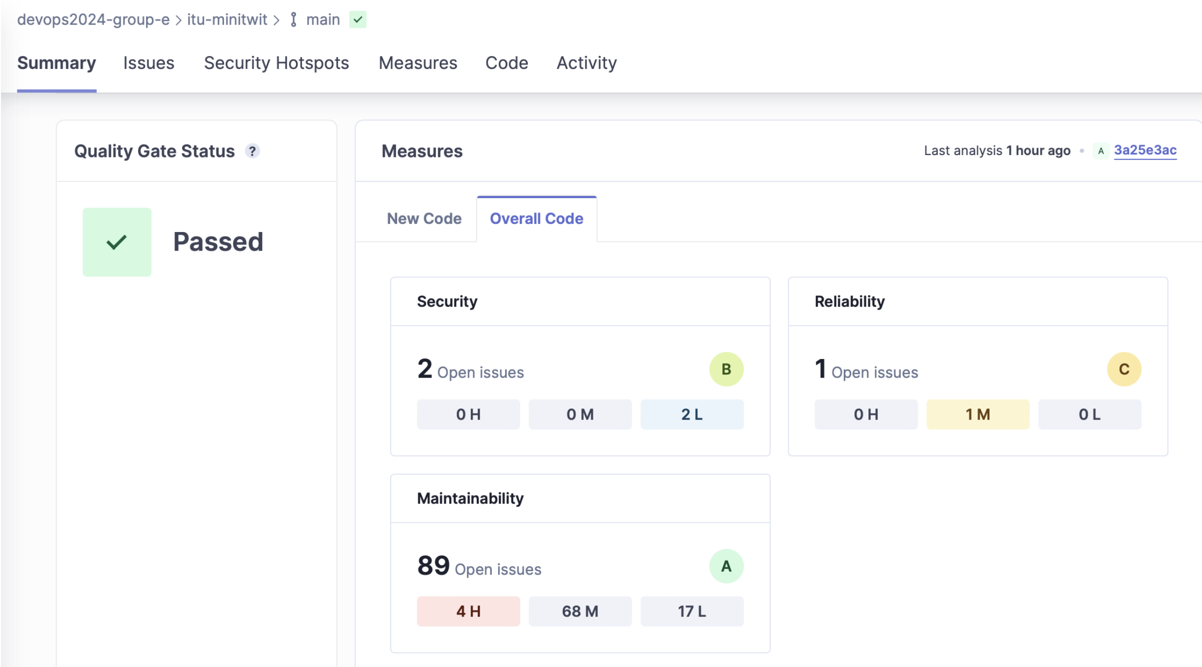

Figure 8: *Sonarcloud code analysis.*

C Decision Log

C.1 Language: C#

We decided to use C# for refactorization. We did this due to C# being object-oriented and a stable language. C# is used by many companies and is therefore known to many. Because C# is is used by many it also exists good documentation easily accessible. There exist both micro web-frameworks and more advanced front-end frameworks for C# in case we want to develop the application further. C# is also a part of the .NET platform, which allows for language interoperability and access to the Core framework used for web application.

C.2 Micro web-framework: Razor pages

Initially we decided to work with Scriban as the micro web framework, as opposed to e.g. razor pages, because the refactorization from the original version of minitwit seemed more direct in Scriban. This is due to the fact that Scriban works with .html templates, like we have in the original version, whereas razor pages integrates C# in html, which would lead to more “merging” of code. Scribans parser and render is also both faster and uses less memory than other templates. After trying to find information on Scriban, the group had trouble figuring out how to set it up properly. Comparing the amount of documentation for Scriban versus Razor pages, we then decided to switch the micro web-framework to Razor pages, as this would allow us to start working with the material faster. Razor pages also has simple context of structure and is flexible to fit with any application. Another reason for choosing Razor pages is that each page is self-contained, with the view and code organized together. This makes Razor pages more organized and easier to work with. With the documentation, flexibility and the organization, Razor pages felt like a good choice for a micro web-framework.

C.3 Work log: Notion

To start of with, we decided to keep our work log/diary in Github, since this automatically provides us with useful information, e.g. dates and authors. Also the work log will follow the code completely, and we can see which code belongs with what log points. However, we realized after a few weeks that this didn’t work as intended. The reason was that the action of making a pull request and requesting a review after just a tiny addition to the log seemed too complicated. As a consequence we ended up keeping our log in Notion and then moving it to GitHub afterwards, which was just double the work and definitely not the intention from the start. Therefore, we decided to move the log to Notion, which also has the advantage that everyone can follow each others writing in real time and collaborate in writing simultaneously.

C.4 Database connection: Entity Framework

We have chosen entity framework for our database connection as it works well with .NET. Entity Framework has the possibility of generating a model from an already existing database. Furthermore, it lets us load data using C# classes and it supports LINQ. Also it works well with SQLite.

C.5 Server host: Digital Ocean

We have chosen Digital Ocean as our cloud hosting service, as it is a cheap cloud hosting service that allows for developers to have quite a lot of control over their server hosting. Furthermore, a lot of popular cloud hosting alternatives have a tendency to be more expensive (typically not by a lot) and with this increased expense they have a lot of options, which can also be configured. As some of the students in the group have limited experience working with cloud servers, we believed Digital Ocean would be a good choice, as it is free, and there are a lot of guides and good documentation.

C.6 CI/CD: GitHub actions

For CI/CD we chose GitHub actions. The reason is that we are already using the GitHub platform and that it is well documented. Furthermore, it is easy to configure secrets and refer to them from the .yaml file. Also, it collaborates nicely with Digital ocean which hosts our server.

C.7 Database: Sqlite to Postgres

We have chosen to use postgres when we had to start create our dockerfile. We looked for a microsoft or another official docker image from a good source(that is from a big known company, that we can trust). But we couldn't find one. We saw that Helge used something he created himself. So we had to choose among some of the databases that has been containerized. So that was the first two requirements. The third requirement is that we need relational database. This remove well known document databases like MongoDB. At the end we had two good options. Microsoft SQL and Postgres.

Feature wise they offer the same things. They are both good at handling concurrent requests and have transactions. They both offer the possibility to add indexes, and to have multiple servers act as a cluster in order to handle even more requests.

We chose the solution that we have had experience with before. Both in our previous course *Introduction to Database Systems*, and in previous projects done at ITU. We have had SOOO many new technologies. So it felt almost relieving to not use a new one again.

C.8 Monitoring: Grafana

Looking for a tool to visualize the metrics of dotnet applications and maybe other tools in our tech stack as well required a versatile tool. That is, we needed to have a tool that we could search and visualize the current status of our application. Obviously a lot of tools can handle this. For that reason it is nice to see that 'Open Telemetry' is announced and widely used. Both as a collector and as an exporter.

So a hard requirement is that the monitoring system has to be able to integrate with Open Telemetry in order to make it easier to change monitoring system if required.

Also we would prefer to have a open source, free of charge product that is being maintained.

C.9 Metrics collector: Prometheus

There has not been that much thought process about Prometheus. It is open source and can collaborate with the open telemetry exporter. Also it is able to connect to the docker metrics as well.

C.10 Database migration: Digital ocean

We have considered hosting a separate server on digital ocean, and using digital oceans own database service. We considered digital oceans options, due to the wide array of databases that they support. However, when researching about trouble shooting, and problems using this solution, we found very few answers from crowd sourced platforms such as stackoverflow, and a lot of guides from the digital ocean docs themselves.

C.11 Logging stack: Grafana + Loki + Promtail

We believe that the amount of applications used can get very bloated. Therefore, we have decided to use Grafana Loki, as we have already setup Grafana last week when implementing monitoring.

C.12 Configuration management: Ansible

We realise now that we need a tool that easily manages the desired state of our running VM's. Especially now that we are going to create more and that we have to move our infrastructure before the 19th of April to another DigitalOcean account.

For that reason we need a simple way to have a configuration of our VM's defined that manages our provisioning. Our requirements are the following:

- Simple setup

- Should be pushable i.e. as soon as we have changed the configuration then it should push the desired state out to our servers
- Keeps track of it already had been running a specific portion of the provisioning scripts
- Does not require us to learn a new language

We have been looking at different players in the market. Especially we have been looking at the CNCF and asking colleagues. Here are some of the products we looked at:

- Ansible
- Fabric - Not really a configuration management system but a “simple” push based script system
- Chef

C.12.1 Fabric

Fabric seemed nice, however it lacked some of the basic things that we wanted to solve, which was the idempotence of the desired state of the VM. This we would have to script ourselves. Also we would have to add another language to our repo, that is python, which would require us to again setup our development container to have both dotnet and python. Our experience with that was that it was very hard to get right, and we finally have a setup now that works on all of our laptops. So we didn’t want to break that now.

C.12.2 Chef

Chef is a configuration management system, just like Ansible as well. It provides a hybrid solution of pull and push based. That is, we push the configuration management changes to a central server which and then the VMs will pull the changes from the server (probably after some interval).

As can be seen this requires also that the VM’s has the Chef Client installed and configured to point at the Chef Server. This provides obviously the nice feature that chef server do not have to know which servers to configure. It is up to the Chef clients to know where to pull the configurations from.

The downside of this setup is the complexity. Seems like a lot of things that needs to be taken care of. And the fact that we would have a load overhead by adding clients to our small web server VM’s would maybe

In order to create a configuration we would have to create what they call cookbooks. This is written in Ruby as i can see. The nice thing about this is that there are testing frameworks implemented. This provides a nice way to be sure that the configuration should work can configure as we intend it. However, playing with Vagrant it has become obvious that our group does not have a lot of knowledge in Ruby and we would currently rather invest time in becoming better at the observability tools that we have rather than learning a new language.

From a Security point of view it uses some of the protocols that we have already opened for. That is it uses standard HTTP protocol to communicate between the server and nodes, and authorizes through certificates.

C.12.3 Ansible

Some of the nice features with Ansible is that the setup seems very simple.

- One server to distribute to a list of servers
- No additional clients to install
- No code for defining configurations, it uses a yaml to define the configuration of the VMs

Obviously the downside is that there is a central server that needs to have a list of the created VM's which then needs to be maintained. Furthermore, from a security perspective we would have to allow ssh and that the ansible server would have to store the private ssh key to access all of the servers. Also we have a hard time trying to understand how we are supposed to push this to the ansible server and then run it? Would that be another ssh session?

Looking a bit further into it is seems like the way to automate the configurations of the server can be done with a proprietary product called Ansible Automation, which requires a license. But some has been able to create a custom github action that can push these changes to the servers. This requires that the ssh port is open to the public, which it already is.

As with Chef, Ansible also has their own term for a configuration, i.e. a Playbook. The term is from American sports, where each task in the playbook is a play that constitutes the playbook. Each play in the playbook is idempotent. That is, it seems to provide simple idempotence, which is fine in most cases, but it some cases it may require some more advanced idempotence.

C.12.4 Why do we choose Ansible?

We chose ansible because the setup seems simple. It provides the features that we need i.e. a nice and easy setup that does not require a new client to be installed on our application servers. Configurations described in yaml and not in some other language as ruby and python.

C.13 Adding E2E and integration tests early

We had early on been handed some integrations test and E2E tests for namely the frontend and Simulator API. This was very nice when we had to develop our system to begin with. But we quickly realised that in order to be comfortable and to ensure that our future changes in the software did not break our interface with the simulator then we had to add them to our CI pipeline via github workflows.

However, we again realised that having them in python would make it harder for us to extend our tests or create more tests as we progress further in our maintenance cycle that we needed to refactor it to C#.

C.14 Infrastructure as Code (IaC): Pulumi

We have lately been a bit frustrated at Vagrant. It seems like it lacks some collaborative features that makes it hard to work with. Also, trying to make it collaborate with a configuration management tool like Ansible proved to be very hard and very time consuming considering that it should "just be plug and play". So we didn't make that work either. So to eliminate a pain that we currently have, we are looking towards encoding our infrastructure as code now. What is our requirements?

- Should be easy to work with
  - That is, we would prefer to keep the configuration in **Yaml** or **C#** as we do not want to increase complexity of learning a new language again
- Can use it in a **Github Actions Workflow** such that we can create changes in the minitwit repo, create a PR, and then push changes in production when everything is working
- This is not a hard requirement, but it would be nice if we can use an **Open Source** tool.
- A tool which can work for multiple Cloud Providers, such that we can change provider if we want to. But since we are at digitalocean(which we are pretty happy with so far) then we want them to support DigitalOcean.

We looked at the following options:

- Vagrant - Is it still a good option or not?
- Terraform - (As stated in a later lecture is something that we are going to look at)
- Pulumi - A tool we stumbled upon in a DigitalOcean video
- OpenTofu
- Write code ourselves with a custom client to DigitalOcean

C.14.1 Vagrant

Is a very nice tool for creating infrastructure quickly. What we have experienced so far is that it is hard to collaborate with. It saves metadata files on the local machine to sort of keep some state. It only pushes the creators ssh key to the server, and not the other ssh key, making us copy paste ssh keys to the server manually. Last but not least it is written in a language that none of us have experience with(Ruby). There is not any github action for using Vagrant, which we think has something to do with the way that it handles its state. Looking at Vagrants comparison between Vagrant and Terraform(which is also a tool made by hashicorp) the following is stated:

Vagrant is a tool for managing development environments and Terraform is a tool for building infrastructure... and more features provided by Vagrant to ease development environment usage... Vagrant is for development environments. https://developer.hashicorp.com/vagrant/intro/vs/terraform

By this description Hashicorp haven't intended to build Vagrant for the scenario that we use it for today. That is, they do not and will not improve the tool in the direction that we intend it to be. Furthermore, it does not make sense for use to keep Vagrant to setup a local development environment because we already have docker and docker compose that takes perfectly care of that.

C.14.2 Terraform

As we can see in the above section, Terraform is a product that is used for the specific functionality that we want to. It can integrate with our github workflow. It does so with Terraform cloud to keep the state of the infrastructure. It can create infrastructure on multiple cloud vendors. It also has support for DigitalOcean.

One of the downsides is that they introduce their own language HCL. This is something that we do not want to have. We want to limit the amount of languages that our repo contain to a minimum. Although we can see that they actually have what they call CDK (Cloud development kit) for Terraform. But we doubt that is their first priority to maintain. The nice thing about it is that it can add testing to Terraform.

Also Terraform is not open source, which makes it hard to trust the tool. A quick search on the internet is that others have forked a former repo of Terraform and created a Terraform look a like(OpenTofu) which reveals that the community do not trust Hashicorps development of the tool now that they have transitioned from Open Source to a BSL (Business Source License).

C.14.3 Pulumi

Pulumi is similar to Terraform, in the way that it also provides a cloud solution to keep state of the infrastructure. It does not offer a new fancy language to write the infrastructure, but instead offers SDK's in languages that we already know. Both Pulumi and Terraform follows what is called Desired State Infrastructure. That is they try to keep the infrastructure in the desired state that the code describes.

It contains support for CI / CD tools. So we can integrate it into our existing workflows. In contrast to Terraform it is open source. It seems to be widely adopted and is known for being easy for developers to get started.

C.14.4 OpenTofu

OpenTofu is a forked version of Terraform. It keeps the structure in HCL. What makes this interesting is that the community has actively taken a decision to maintain this project in order for it not be a BSL tool, distancing them from Hashicorps decision with Terraform. This can be seen in the OpenTofu Manifesto https://opentofu.org/manifesto/. The nice thing about OpenTofu is that it has later been adopted in the Linux Foundation and for sure here to stay.

As it is a copy of Terraform it has some of the same nice features and downsides. This can be seen by having their own custom language again to describe the infrastructure called OpenTofu Language. OpenTofu has not done the same thing as Terraform in terms of create CDK's. That means the only option is the OpenTofu Language.

The downside is that we are unsure whether there is support for digitalocean as a provider. There has to be, but it is not clear and we will need to spend time and look into it.

C.14.5 Write C# code with DigitalOcean client

This option may not be the best option. Compared to the others we will have a hard coupling to DigitalOcean. This will make it hard for use to change provider in the future if it is required. This will also require us to keep an eye out for the changes to the general api and make adjustments to the custom client that we would have to make. We could also find a client that is made for us written in C#, but then we would have to find one that is being maintained.

The realisation that we have here is that the maintenance of just written code for the infrastructure becomes a project in it self. We would have to manage a lot of other dependencies, which is taken care of from the other projects. Also we do not have any testing capabilities as the other tools have.

Also we will not benefit from things that others may have created. A large community for an open source tool allows for sharing solutions and code for that specific tool. We do not really have that option here, as we are going to create a complete new tool with this option.

C.15 Scaling: Docker swarm

We need to add some sort of horizontal scaling to our system.

Our requirements

- Opensource
- Works well with ansible/docker/pulumi.

C.15.1 Docker swarm

[1]

- Opensource
- It is based on the Docker API, and therefore is **lightweight**, and **easy to use**.
- Docker swarm allows for limited scaling, and isn't recommended for big systems, so if minitwit ever became a bigger service, this could potentially become a problem.
- It is easy to add new nodes to existing clusters as both manager and worker
- Compared to other tools like kubernetes
- Simple to install compared to Kubernetes
- Same common language as we're already using to navigate in the structure
- Limited guides on interactions between docker swarm and ansible.

C.15.2 Building our own redundant load balancer:

- Time consuming to create
- Less information on how to build it
- Probably harder to add new nodes than for docker swarm

C.15.3 Kubernetes

- Takes more planning to implement than docker swarm
- Has a quite steep learning curve
- It can sustain and manage large architectures and complex workloads.
- Has a GUI

C.16 Upgrade/Update Strategy: Rolling updates

Rolling updates is the default update strategy in Docker Swarm and therefore the one we have chosen so as to not make the implementation more complicated and "home-made" than it needs to be.

C.17 Precommit

D Log

E Security Assessment

E.1 Risk Identification

E.1.1 Asset Identification

By browsing through our setup and documentation, we identified the following list of assets:

- Web application
- Public GitHub repository
- Digital Ocean servers
- Tools
  - Ansible
  - Code Climate
  - Docker, Docker Compose & Docker Swarm
  - GitHub Actions
  - Grafana
  - Linters
  - Loki
  - Pulumi
  - Sonar Cloud

E.1.2 Threat Source Identification

To help us identify possible threats to the system, we have consulted the OWASP *Top 10 Web Application Security Risks*[4], which describes the following threats:

1. Broken Access Control
2. Cryptographic failures
3. Injection attacks
4. Insecure Design
5. Security misconfiguration
6. Vulnerable and outdated components
7. Identification and authentication failures
8. Software and Data integrity failures

9. Security Logging and Monitoring Failures
10. Server Side Request Forgery

E.1.3 Risk Scenario Construction

Based on the information gathered from the two previous steps, we have constructed the following risk scenarios and outlined which of the OWASP top 10 risks affecting the scenario:

1. **URL Tampering**

   The attacker can construct URL in the **web application** with user ID such that they bypass login and are able to write message from another user's account.

   This would be an issue of *Broken Access Control* and *Server Side Request Forgery*.

2. **Log Injection**

   The attacker can fabricate log information via injection attack in the **web application** as a means to hide their activity and ill-intentioned actions.

   This would be an issue of *Security Logging and Monitoring Failures* and *Injection attacks*.

3. **Password Brute Forcing**

   The attacker can brute force login credentials in the **web application** by taking advantage of no timeouts and weaker hash implementation.

   This would be an issue of *Cryptographic failures* and *Identification and authentication failures*.

4. **Depricated Dependencies**

   The attacker can identify weak, outdated or depricated tools and dependencies in the system's CI/CD pipeline via **GitHub Actions**, which is publicly availble through the **GitHub repository**.

   This would be an issue of *Vulnerable and outdated components* and *Software and Data integrity failures*.

5. **Open Ports**

   The attacker can scan the public IP addresses of the **Digital Ocean** servers to find unnecessarily or unexpectedly open ports with known vulnerabilities, which can be exploited in further attacks.

   This would be an issue of *Security misconfiguration*.

6. **SQL Injection**

   The attacker can target the login page of the **web application** with SQL-injection attacks to strike the database.

   This would be an issue of *Injection attacks*.

7. **Exposed Secrets**

   The attacker can get access to the system or associated tools via secrets found written in the code in the public **GitHub repository**.

   This would be an issue of *Identification and authentication failures* and *Security misconfiguration*.

E.2 Risk Analysis

E.2.1 Likelihood Analysis

Likelihooed will be graded on the following scale:{Rare, Unlikely, Possible, Likely, Certain} with *Rare* being the least severe and *Certain* being the most severe.

1. **URL Tampering**

   We examined the different URLs on the web application and couldn't find any where the ID's or login parameters were exposed. Therefore we deemed the likelihood to be **Unlikely**.

2. **Log Injection**

   We got warnings from Sonar Cloud that we had logging vulnerabilities in our code several different places. Therefore we deemed the likelihood to be **Likely**.

3. **Password Brute Forcing**

   We currently don't have any measures in place to combat brute force attacks and no password requirements for the users when signing up. Therefore we deemed the likelihood to be **Possible**.

4. **Depricated Dependencies**

   Our GitHub repository and thereby our workflows for GitHub Action as well are public for anyone to see. We are in the workflows using templates of actions made available and written by other alongside showcasing some of various tools we use. We currently rely on the fact that the actions and tools we use are secure, but haven't incoporated anything that checks whether that is true. However, many of the tools we use are well-known and therefore we hope that known vulnerabilities are getting discovered rather quickly. Therefore we deemed the likelihood to be **Possible**.

5. **Open Ports**

   All of the IP addresses for our servers are public in Digital Ocean, which makes it very easy to scan for vulnerabilities. We have put up firewalls and have taken measures to only have necessary ports open, but are also aware that accidental port exposure, through some of the tools we are using, is possible. We are convinced that this is how our server got hacked during the course. Therefore we deemed the likelihood to be **Certain**.

6. **SQL Injection**

   We have made sure to sanitize user input on the login page of the web application. Therefore we deemed the likelihood to be **Rare**.

7. **Exposed Secrets**

   We have been conscious to ensure that secrets were either kept locally where only ourselves can access them such as secret keys for logging into the servers or used the 'environment secrets' tool on GitHub if secrets had to be accessed from the repository. Additionally, we have not made generic passwords, but rather used random password generators to get stronger passwords. Therefore we deemed the likelihood to be **Rare**.

E.2.2 Impact Analysis

Impact will be graded on the following scale:{Insignificant, Negligible, Marginal, Critical, Catastrophic} with *Insignificant* being the least severe and *Catastrophic* being the most severe.

1. **URL Tampering**

   This would breach both the confidentiality and the integrity of the system's security. We still keep all our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

2. **Log Injection**

   This could be used to disguise an attackers activity and attack attempts on the web application. However, it wouldn't give them access to the server or applicition itself nor other data than the logs. We would still want to know if someone was trying to attack our system. Therefore we deemed the impact to be **Marginal**.

3. **Password Brute Forcing**

   This would breach both confidentiality and integrity. In very severe cases, it could also affect the availability, if the requests to login became to intense. We would still keep all of our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

4. **Depricated Dependencies**

   This would have a very big attack surface, as we most likely wouldn't know which tool or where in the application process we could have a vulnerability. The target for a vulnerability could thereby vary in severity, but could in the worst case have severe consequences. Therefore we deemed the impact to be **Catastrophic**.

5. **Open Ports**

   This again would have a big attack surface, as we don't know which tool or where in the application we potentially could have a vulnerability exposing ports for an adversary to attack. If the attacker ended up gaining access to our servers, they would have full control over the application in the worst case. Therefore we deemed the impact to be **Catastrophic**.

6. **SQL Injection**

   This would breach both confidentiality and integrity. The data could both be compromised and lost, but we would have a backup of the database. Therefore we deemed the impact to be **Critical**.

7. **Exposed Secrets**

   If any of the secret in the GitHub repository were to fall into an attackers hands, they would be able to get access to our setup and thereby dismantle the entire system. Therefore we deemed the impact to be **Catastrophic**.

E.2.3 Risk Matrix

Based on the points made about the likelihood and impact of each risk scenario, we have constructed the following risk matrix to indicate the severities and prioritize the scenarios:

	Rare	**Unlikely**	**Possible**	**Likely**	**Certain**
Catastrophic	Exposed Secrets		Depricated Dependencies		Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing		
Marginal				Log Injection	
Negligible					
Insignificant					

Figure 9: *Risk matrix based on security assessment*

E.2.4 Action Plan

We discussed the results of the security assessment and decided on focusing on the risk scenarios placed in the red area of the risk matrix.

For the open ports, we went through our firewalls setting and scanned our main server's IP address to see the current open ports. We tried to close the port we had open for Prometheus, but got conflicting results, when we checked the firewall istelf compared to the scan of the IP address.

For the log injection, we went through our code and ensured that all user data was sanitized before added to the log, such that it couldn't be compromised with.

For the depricated dependencies, we added the tool *dependabot* to our CI/CD pipeline, which makes sure that the dependencies used in our system are up to date and thereby less vulnerable to older known exploits.

For the password brute forcing, we decided to leave it as it, due to the group not having enough information on how the creation of users and login works in the simulator and thereby not knowing if the simulator could handle password restraints or 2-factor authentication, which would have been our solution to improve this issue.